Parallel Inclusion-based Points-to Analysis

Mario Méndez-Lojo[1] Augustine Mathew[2] Keshav Pingali[1,2]

[1]Institute for Computational Engineering and Sciences, University of Texas, Austin, TX

[2] Dept. of Computer Science. University of Texas, Austin, TX

marioml@ices.utexas.edu, amathew@cs.utexas.edu, pingali@cs.utexas.edu

Abstract

Inclusion-based points-to analysis provides a good trade-off between precision of results and speed of analysis, and it has been incorporated into several production compilers including gcc. There is an extensive literature on how to speed up this algorithm using heuristics such as detecting and collapsing cycles of pointer-equivalent variables. This paper describes a complementary approach based on exploiting parallelism. Our implementation exploits two key insights. First, we show that inclusion-based points-to analysis can be formulated entirely in terms of graphs and graph rewrite rules. This exposes the amorphous data-parallelism in this algorithm and makes it easier to develop a parallel implementation. Second, we show that this graph-theoretic formulation reveals certain key properties of the algorithm that can be exploited to obtain an efficient parallel implementation. Our parallel implementation achieves a scaling of up to 3x on a 8-core machine for a suite of ten large C programs. For all but the smallest benchmarks, the parallel analysis outperforms a state-of-the-art, highly optimized, serial implementation of the same algorithm. To the best of our knowledge, this is the first parallel implementation of a points-to analysis.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

General Terms Algorithms, Languages, Performance

Keywords Inclusion-based Points-to Analysis, Irregular Programs, Amorphous Data-parallelism, Galois System, Synchronization Overheads, Extensive Transformers, Iteration Coalescing, Binary Decision Diagrams.

1. Introduction

Points-to analysis is a static analysis technique that determines what a pointer variable may point to during the execution of a program. The results of this analysis are useful for program optimization, program verification, debugging and whole program comprehension [13]. In the literature, there are many variations of points-to analysis: some analyses are context-sensitive while others are context-insensitive, some are flow-sensitive while others are flow-insensitive, etc. [3, 4, 7, 8, 10, 32, 35]. These variations make different trade-offs between precision and running time, but production compilers like gcc and LLVM [21] seem to have settled on context-insensitive, flow-insensitive points-to analysis because more precise alternatives are currently intractable for large programs.

The most popular algorithm for context-insensitive, flow-insensitive points-to analysis is known as inclusion-based or Andersen-style analysis [3]. As explained in Section 2, it requires the solution of a system of set constraints derived from the program. The asymptotic worst-case complexity of the solution procedure is $O(n^3)$, where n is the number of variables in the program. Although this worst-case behavior is not usually observed in practice, the solution procedure is sufficiently expensive that much research has gone into heuristics for speeding it up. A key observation was made by Fahndrich *et al* [7], who showed that detecting and eliminating cyclic constraints can make a big difference in the running time because it reduces the size of the constraint system. Since cyclic constraints can arise dynamically during the solution of the constraint system, it is not sufficient to preprocess the system of constraints *offline* before the constraint system is solved [29]. A recent advance was made by Hardekopf and Lin who invented ingenious heuristics for determining when to trigger cycle detection *online* during constraint solving [8].

The advent of multicore processors that support a shared-memory programming model opens up a new avenue for speeding up these kinds of expensive static analysis algorithms. Unfortunately, most work on parallel programming has focused on high-performance applications in which the key computations are matrix and vector operations such as linear system solvers and FFTs; as a consequence, little is known about how to parallelize "irregular" algorithms like inclusion-based points-to analysis in which the key data structures are graphs, sets, binary decision diagrams [5] (BDD's), etc. There is an enormous literature on static analysis techniques for finding parallelism in programs [15], but these techniques fail to find much parallelism in most irregular algorithms. This is because dependences between computations in irregular algorithms are functions of runtime values, so static parallelization techniques end up being overly conservative.

In the literature, several papers mention the possibility of parallelizing points-to analysis, but there are no implementations or evaluations[1]. These papers use partitioning of the analysis problem to reduce memory consumption [30] or to perform different kinds of points-to analysis on different partitions [14, 36], and suggest in passing that partitioning might also be useful for parallel implementations of points-to analysis. Pereira *et al* [25] discuss the parallelization of their "wavescalar" approach, but they do not have an implementation. It is not clear to us that partitioning is a useful approach for inclusion-based points-to analysis; as we mentioned before, a key step in speeding up inclusion-based points-to analysis is the detection and collapsing of certain cyclic constraints that arise dynamically during the constraint solving process, and these cycles may cross partitions. In any case, partitioning is complementary to the approach presented here.

This paper describes the first parallel implementation of inclusion-based points-to analysis and it makes the following contributions.

1. In Section 3, we show that inclusion-based points-to analysis can be formulated entirely in terms of graphs and graph rewrite rules. The system of set constraints is expressed as a graph, and the solution to the set constraints is expressed in terms of rewrite rules for this graph. This formulation exposes the *amorphous data-parallelism* [26] in this algorithm, permitting the derivation of a parallel implementation of inclusion-based points-to analysis. As in most irregular algorithms, dependences in this algorithm are functions of runtime data, so it is necessary to use optimistic or speculative parallelization. Our implementation is based on the Galois system [1], described in Section 4.
2. Speculative parallel execution of programs can incur substantial runtime overheads. Recent work has shown algorithmic structure must be exploited to eliminate this overhead [23]. In Section 5, we show that the graph-theoretic formulation of inclusion-based points-to analysis reveals important and novel algorithmic structure that can be exploited to dramatically reduce the overheads of speculative parallel execution of this algorithm.
3. Although online cycle detection is fundamental for achieving good performance, its addition to the basic algorithm represents a challenge for parallelization. In Section 6, we show how to reduce the problem of online cycle detection to the simpler problem of merging two nodes in the graph, and then show how to safely interleave merges with graph rewriting rules for constraint solving.

Using these ideas, we implemented a parallel Java version of a state-of-the-art points-to analysis algorithm by Hardekopf and Lin [8] (their implementation has been incorporated into gcc and LLVM). The experimental results in Section 7 show that our parallel implementation scales reasonably well (up to 3x in a 8-core machine) for a benchmark suite of C programs ranging in size from 53,000 to 558,000 variables. Furthermore, this parallel version outperforms the serial reference implementation for all but the smallest benchmarks.

2. Inclusion-based points-to analysis

Inclusion-based points-to analysis is context-insensitive and flow-insensitive. A context-sensitive analysis analyzes a procedure separately for each context in which it is invoked; in contrast, a context-insensitive algorithm would merge information from all call sites. Context-insensitive analyses produce less accurate information, but they run faster than context-sensitive alternatives for most problems. Flow-sensitive analyses produce results that are specific to particular points in the program, while flow-insensitive alternatives do not consider control-flow and produce a single conservative approximation valid for all program points. Hind has written an excellent survey of the enormous literature on different varieties of points-to analysis [13]. The precision of flow and context-insensitive alternatives is sufficient for many application clients, so they have been incorporated into production compilers such as gcc or LLVM.

Context-insensitive, flow-insensitive points-to analyses are either unification-based or inclusion-based. If a and b are pointer variables, and $a = b$ is an assignment statement in the program, a unification-based algorithm will assert conservatively that the points-to sets of a and b are equal; in contrast, an inclusion-based algorithm makes the minimal deduction that the points-to set of b must be a subset of the points-to set of a. The points-to analyzers present in gcc and LLVM are inclusion-based.

2.1 Set constraint formulation

Inclusion-based pointer analysis is usually formulated as a set-constraint problem. A single pass through the program code generates a system of set constraints that implicitly defines the points-to set $pts(v)$ for each variable v in the program. Nested pointer dereferences are eliminated by introducing auxiliary variables, leaving only one pointer dereference per statement. Figure 1 shows the different types of assignment statements and the set constraint generated for

[1] We are grateful to Ben Hardekopf for this information, and for the citations in this paragraph.

Statement	**Name**	**Constraint**
$a = \&b$	pointer	$loc(b) \in pts(a)$
$a = b$	copy	$pts(a) \supseteq pts(b)$
$a = *b$	load	$\forall v \in pts(b) : pts(a) \supseteq pts(v)$
$*a = b$	store	$\forall v \in pts(a) : pts(v) \supseteq pts(b)$

Figure 1. Formulating set constraints

Program	**Constraints**
$a = \&v;$	$loc(v) \in pts(a)$
$*a = b;$	$\forall v \in pts(a) : pts(v) \supseteq pts(b)$
$b = x;$	$pts(b) \supseteq pts(x)$
$x = \&w$	$loc(w) \in pts(x)$

Figure 2. Running example

each type. In these constraints, $loc(v)$ represents the memory location denoted by v. The most complex constraints result from variable dereferencing in load and store statements. For a load statement, the constraint asserts that for every variable v in the points-to set of b, the points-to set of v is a subset of the points-to set of a. The constraint for a store statement asserts that for every variable v in the points-to set of a, the points-to set of b is a subset of the points-to set of v. Load and store constraints are called *complex* constraints. Figure 2 shows a simple program and its associated system of set constraints.

Systems of set constraints such as the one in Figure 2 are higher order constraint systems, so one cannot appeal to the usual monotonicity theorems [2] to assert that these systems have solutions. For a given program P in which the set of variables is V, let a *points-to assignment* be a function $V \to 2^V$ (intuitively, it is a function that maps each variable to the subset of variables it points to). Points-to assignments for a given program can be partially ordered in a natural way: if A_1 and A_2 are points-to assignments, then $A_1 \leq A_2$ if for all variables v, $A_1(v) \subseteq A_2(v)$ (in words, for every variable v, the points-to set of v in A_1 is a subset of the corresponding set in A_2). Let $\mathcal{A}$ be the set of points-to assignments for a given program P with variables V. Define

- $A_1 \wedge A_2\ (v) = A_1(v) \cap A_2(v)$
- $A_1 \vee A_2\ (v) = A_1(v) \cup A_2(v)$

Theorem 1. Let $\mathcal{C}$ be a system of set constraints generated from a program P using the rules in Figure 1. $\mathcal{C}$ has a least solution.

Proof. The proof involves the following steps.

- If $\mathcal{A}$ is the set of pointer assignments for P, $(\mathcal{A}, \leq, \wedge, \vee)$ is a finite lattice.
- Let $\top$ denote the largest element of $\mathcal{A}$ (so $\top(v) = V$ for all $v \in V$). $\top$ satisfies $\mathcal{C}$.
- If S_1 and S_2 are two solutions of $\mathcal{C}$, then $S_1 \wedge S_2$ is also a solution.

□

2.2 Solving systems of set constraints

Andersen described a simple iterative algorithm for finding the least solution of these set constraint systems [3]. The points-to sets of all variables are initialized to the empty set, and the constraints are processed iteratively until the points-to sets converge. When a constraint is considered, it is satisfied ”locally” by growing some of the points-to sets as needed. The constraint for a pointer statement can be satisfied by adding $loc(b)$ to $pts(a)$ if it is not in that set. For all other kinds of statements, a constraint of the form $P_1 \supseteq P_2$ is satisfied by setting P_1 to $P_1 \cup P_2$. Making these kinds of updates to satisfy one constraint may violate others considered previously, so it is necessary to iterate over the system of constraints repeatedly until convergence.

To avoid redundant evaluations, implementations of this iterative algorithm construct a *propagation graph* in which there is a node for each variable, labeled with its points-to set; these labels are initialized to the empty set. Pointer constraints are processed by adding $loc(b)$ to the label of a (we will use the notation $b \in pts(a)$). Each copy constraint $pts(a) \supseteq pts(b)$ is processed by adding an edge $b \to a$; intuitively, any element added to the points-to set (label) of b will ”flow down” into the points-to set of a. Complex constraints are not explicitly represented in the graph; they are maintained in a separate list. A node b is processed in two steps:

1. For each outgoing edge $b \to a$, check whether $pts(a) \supseteq pts(b)$ holds. If not, propagate $pts(b)$ to node a, i.e., $pts(a) := pts(a) \cup pts(b)$.
2. For each $v \in pts(b)$:
   - for each load constraint $a = *b$, add an edge $v \to a$ to ensure that $pts(a) \supseteq pts(v)$.
   - for each store constraint $*b = a$, add an edge $a \to v$ to ensure that $pts(v) \supseteq pts(a)$.

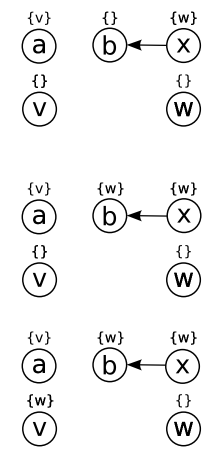
This iterative process is repeated until the points-to sets and graph edges do not change.

Figure 3 shows the iterative constraint solving process for the running example in Figure 2. The points-to set of a variable is shown in braces above that node. In each stage, violated constraints are shaded.

3. Constraint solving by graph rewriting

In this section, we introduce one of the contributions of this paper: we show that inclusion-based points-to analysis can be formulated entirely in terms of graph rewriting rules for certain *constraint graphs* that can be constructed from the program. As we will see later in this section, there are just three rewrite rules, and although they can be applied in any arbitrary order, the final graph is unique (for a given initial graph). The solution to the inclusion-based points-to analysis problem can be read off from the final constraint graph.

Propagation graph



Constraint system

$$loc(v) \in pts(a)$$
$$\forall v \in pts(a) : pts(v) \supseteq pts(b)$$
$$pts(b) \supseteq pts(x)$$
$$loc(w) \in pts(x)$$

$$loc(v) \in pts(a)$$
$$\forall v \in pts(a) : pts(v) \supseteq pts(b)$$
$$pts(b) \supseteq pts(x)$$
$$loc(w) \in pts(x)$$

$$loc(v) \in pts(a)$$
$$\forall v \in pts(a) : pts(v) \supseteq pts(b)$$
$$pts(b) \supseteq pts(x)$$
$$loc(w) \in pts(x)$$

Figure 3. Solving the constraint system for the running example

The major advantage of this graph rewriting formulation is that it simplifies the reasoning about these kinds of analysis algorithms - in particular, about their parallelization. The other benefit is that it enables us to take advantage of existing optimization technology and tools that were developed by the Galois project for other irregular problems, as we will show in Section 4.

3.1 Constraint graphs

Given a program with the four kinds of statements shown in Figure 1, the constraint graph for that program contains (i) one node for each variable in the program, and (ii) one edge $a \to b$ for each type of statement involving variables a and b, labeled with the type of that statement, as shown in Figure 4. Note that there might be many edges from a node a to a node b, but they must have different labels[2]. Figure 5 shows the constraint graph for the running example.

More formally, a constraint graph for a program P is a graph (V, E) in which there is one node in V for each variable in the program P, and $E = E^p \cup E^c \cup E^l \cup E^s$, where these sets of edges are defined as follows.

- E^p: These are *points* edges. There is an edge $a \xrightarrow{p} b$ in the constraint graph if $a = \&b$ is a statement in program P.
- E^c: These are *copy* edges. There is an edge $b \xrightarrow{c} a$ in the constraint graph if $a = b$ is a statement in program P, .
- E^l: These are *load* edges. There is an edge $b \xrightarrow{l} a$ in the constraint graph if $a = *b$ is a statement in program P,h.
- E^s: These are *store* edges. There is an edge $a \xrightarrow{s} b$ in the constraint graph if $*a = b$ is a statement in program P, .

[2] To be consistent with the standard definition of a graph, we should replace these edges with a single edge with multiple labels, but we have not done this, to keep the description more intuitive.

Code	Name	Edge
a=&b	points	$a \xrightarrow{p} b$
a=b	copy	$a \xleftarrow{c} b$
a=*b	load	$a \xleftarrow{l} b$
*a=b	store	$a \xrightarrow{s} b$

Figure 4. Basic edge types

Program

$a = \&v;$
$*a = b;$
$b = x;$
$x = \&w$

Constraint Graph

$$\begin{tikzcd} a \arrow[r, "s"] \arrow[d, "p"] & b & x \arrow[l, "c"'] \arrow[d, "p"] \\ v & & w \end{tikzcd}$$

Figure 5. Constraint graph for running example

As we will see next, *points* and *copy* edges are added dynamically to the graph during the graph rewriting process.

3.2 Graph rewriting

There are three graph rewrite rules as shown in Figure 6. Each rewrite rule enforces a certain invariant in the constraint graph by adding edges to the graph as needed. As in general term rewriting systems, these graph rewrite rules can be considered as purely syntactic rules, but it is useful to understand the intuition behind them.

- *Copy rule*: Edge $b \xrightarrow{c} a$ represents the constraint $pts(a) \supseteq pts(b)$, so if $v \in pts(b)$, v must be in $pts(a)$. Edge $a \xrightarrow{p} v$ is added if it is not already in the graph, to ensure this.
- *Load rule*: Edge $b \xrightarrow{l} a$ represents the constraint $\forall v \in pts(b) : pts(a) \supseteq pts(v)$. As a step towards enforcing this, edge $v \xrightarrow{c} a$ is added if it is not already in the graph. The addition of this copy edge may trigger one or more applications of the copy rule.
- *Store rule*: The intuition behind this rule is similar to the intuition behind the load rule.

Notice that each rewrite rule is triggered if there is a node with two outgoing edges at which the relevant invariant is not satisfied because of a missing edge between the destination nodes of the outgoing edges. Such a node is called an *active* node. In Figure 6, the active node for each rule is shaded. When an active node is processed and an edge is added to the graph, it may cause other nodes to become active. There may be many active nodes in a given constraint graph, a fact that we exploit in the parallel algorithm described in Section 4.

A high level description of the algorithm is the following. The constraint graph is built from the program. At each step,

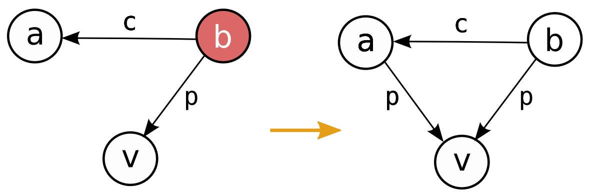
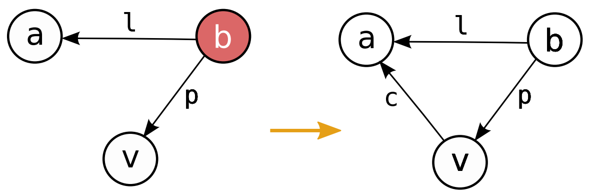
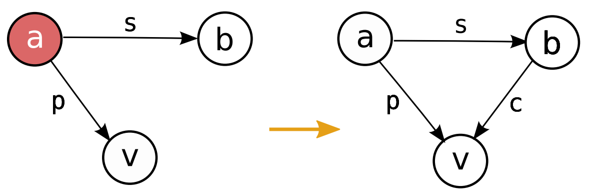
Name	Invariant	Rewrite rule	Ensures
copy	$\exists b \xrightarrow{p} v \wedge \exists b \xrightarrow{c} a$ $\Rightarrow \exists a \xrightarrow{p} v$		$pts(a) \supseteq pts(b)$
load	$\exists b \xrightarrow{p} v \wedge \exists b \xrightarrow{l} a$ $\Rightarrow \exists v \xrightarrow{c} a$		$\forall v \in pts(b)$: $pts(a) \supseteq pts(v)$
store	$\exists a \xrightarrow{p} v \wedge \exists a \xrightarrow{s} b$ $\Rightarrow \exists b \xrightarrow{c} v$		$\forall v \in pts(a)$: $pts(v) \supseteq pts(b)$

Figure 6. Constraint graph rewriting rules

an arbitrary active node is selected and the relevant rewrite rule is applied to the graph. This process can be described formally in terms of pushouts and graph morphisms as is done in graph grammars [6], but we will not do so here. The algorithm terminates when there are no more active nodes in the graph. The solution to the inclusion-based points-to analysis problem can be read off from the final graph: if $v \xrightarrow{p} w$ is an edge in the graph, $w \in pts(v)$. Figure 7 shows this rewriting process applied to the running example.

This high level algorithm can be implemented in the obvious way by keeping a work-list of active nodes. To identify the rewrite rule to be applied, we can store the two outgoing edges of the active node together with that active node in the work-list. When an active node is removed from the work-list, we first check to see if the necessary edge has already been added by a previous step. If so, there is nothing to be done. Otherwise, the relevant edge e is added as required by the rewrite rule, and the source node s of this edge is checked to see if the newly added edge makes s active. If so, it is added to the work-list together with e and its partner edge.

Theorem 2. Given a program P, let $\mathcal{C}$ be the system of set constraints, and let G be the constraint graph. The following facts hold.

(a) The graph rewriting process always terminates.
(b) The final graph produced at the end of the rewriting process is independent of the order in which the rewrite rules are applied.
(c) The solution to the points-to analysis problem read off from the final graph is identical to the least solution of $\mathcal{C}$.

We give a sketch of the proof.

Proof. (a) Follows from the fact that each of the rewrite rules in Figure 6 adds an edge to the graph and does not remove any nodes or edges. In the worst case, the rewriting must terminate when there are *points* and *copy* edges between every pair of nodes.

(b) It is easy to see that the rewrite rules of Figure 6 are locally confluent. From part (a), we know that there are no infinite chains of rewriting steps. From Newman's lemma [16], it follows that the rewriting system is globally confluent and the final constraint graph is unique.

(c) Using an induction on any sequence of rewriting steps, it is easy to show that if A_1 is the points-to assignment read off from the final graph and A_2 is the points-to assignment that is the least solution of $\mathcal{C}$, then $A_1 \leq A_2$. The fact that $A_1 < A_2$ cannot hold can be proved from the fact that when the rewriting stops, there cannot be any active nodes left in the graph.

□

3.3 Parallelism in constraint graph rewriting

In general, there will be many active nodes in a constraint graph at any point during the rewriting process. If the rewriting steps at two active nodes do not interfere with each other, they can obviously be performed in parallel. Therefore, the key problem in parallelizing the graph rewriting process is finding non-interfering active nodes.

As mentioned earlier, this is a far more difficult problem than the well-studied problem of parallelizing *regular* applications like dense matrix multiplication, FFTs and stencil

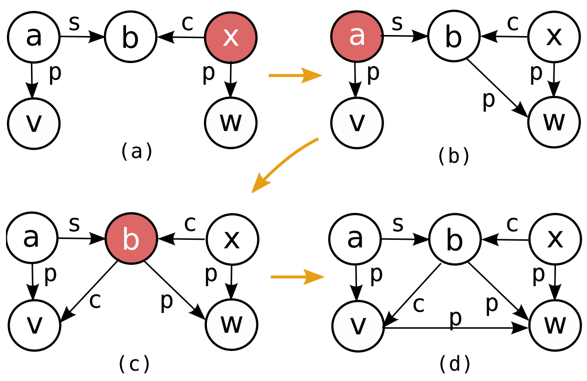


Figure 7. Constraint graph rewrite rule example.

codes [15]. In a regular application, dependences between computations are independent of runtime values, so it is possible in principle to produce a parallel schedule for the application before the program is executed. Unfortunately, inclusion-based points-to analysis is an example of an *irregular* computation in which the computations that need to be performed, as well as the dependences between these computations, are functions of runtime data, so it is not possible to produce a parallel schedule statically.

4. Parallel graph rewriting using the Galois system

One solution to parallelizing constraint graph rewriting is to use the Galois system [1]. To make this paper self-contained, we give a brief description of the Galois system in this section, and show how it can be used to implement parallel graph rewriting.

4.1 Background: the Galois system

The Galois system is intended to support the parallel execution of irregular applications such as those that operate on large graphs and trees. Examples of such applications include n-body simulations, mesh generators, social network applications, SAT solvers, etc.

The abstractions supported by the Galois programming model are derived from an *operator formulation* of irregular algorithms, which we explain using the graph shown in Figure 8. At each point during the execution of such an algorithm, there are certain nodes or edges in the graph where computation might be performed. Performing a computation may require reading or writing other nodes and edges in the graph. The node or edge on which a computation is centered is called an *active element*, and the computation itself is called an *activity*. It is convenient to think of an activity as resulting from the application of an *operator* (graph transformer) to the active node. The set of nodes and edges that are read or written in performing the activity is called the *neighborhood* of that activity. In Figure 8, the filled nodes represent active nodes, and shaded regions represent the neighborhoods of those active nodes. In some

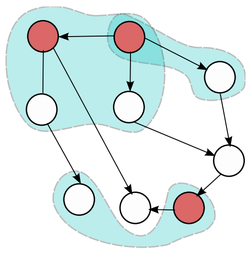


Figure 8. Data-centric view of algorithms.

algorithms, activities may modify the graph structure of the neighborhood by adding or removing graph elements.

In general, there are many active nodes in a graph, so a sequential implementation must pick one of them and perform the appropriate computation. In this discussion, we focus on algorithms such as the preflow-push maxflow algorithm and Delaunay mesh refinement in which the implementation is allowed to pick *any* active node for execution. These are called *unordered algorithms*. Programmers write algorithms in a sequential, object-oriented language like sequential Java, using a *Galois set iterator* to assert that the algorithm is unordered. The Galois set iterator is similar to a set iterator in Java except that it permits new elements to be added to the set while the iterator executes. In addition, the Galois system has a library of concurrent data structures like graphs, priority queues, sets, etc. All concurrency control is implemented within this library.

4.1.1 Baseline parallel execution model

Figure 8 shows how opportunities for exploiting parallelism arise in graph algorithms: if there are many active elements at some point in the computation, each one is a site where a processor can perform computation, subject to neighborhood constraints. In the baseline parallel execution model, the graph is stored in shared-memory, and active nodes are processed by some number of threads. Like thread-level speculation [17] and transactional memory [12], the Galois system uses speculative parallel execution to handle the problem of dependences that can only be elucidated at runtime. A free thread picks an arbitrary active node and speculatively applies the operator to that node, making calls to the graph class API to perform operations on the graph as needed. The neighborhood of an activity can be visualized as a blue inkblot that begins at the active node and spreads incrementally whenever a graph API call is made that touches new nodes or edges in the graph. To ensure that neighborhood constraints are respected, each graph element has an associated exclusive *abstract lock*. Locks are held until the activity terminates. If a lock cannot be acquired because it is already owned by another thread, a conflict is reported to the runtime system, which rolls back one of the conflicting activities. To enable rollback, each graph API method that modifies the graph makes a copy of the data before modification. Like abstract lock manipulation, rollbacks are a service implemented by the library and runtime system. The activity terminates when the application of the operator is complete and all acquired locks are released.

Intuitively, the use of abstract locks ensures that graph API operations from concurrently executing iterations *commute* with each other, ensuring that the iterations appear to execute in some serial order as required by the semantics of the Galois set iterator. There are more sophisticated techniques for checking commutativity, but these are more complex to implement [18]. Commuting graph API operations that touch the same locations in the concrete representation must be synchronized.

The Galois system has been used to parallelize many complex applications including Delaunay mesh generation and refinement, agglomerative clustering, survey propagation, and the preflow-push maxflow algorithm [19].

4.2 Baseline parallelization of constraint graph rewriting

The constraint graph rewriting algorithm described in Section 3 is an unordered algorithm: the active nodes in the graph are the nodes where an invariant of Figure 6 are violated, the operator is the appropriate rewrite rule, and the neighborhood of an activity consists of the three nodes and their incident edges.

Since the baseline implementation uses exclusive abstract locks, a conflict occurs when two or more activities touch the same node. Figure 9 shows two examples of conflicts. In Figure 9(a), one rule wants to add an edge $a \xrightarrow{p} v$, while the other wants to add $x \xrightarrow{p} v$. Because both activities try to lock b and v, they cannot proceed in parallel. A conflict arises in Figure 9(b) because the two activities are trying to add points edges to node a (the destination nodes of these edges are different).

For this algorithm, the baseline parallelization approach adds substantial overheads to a small computation (two edge reads, one edge addition). These overheads come from four sources:

- *Enforcing neighborhood constraints*: Acquiring and releasing abstract locks on neighborhood elements can be a major source of overhead.
- *Copying data for rollbacks*: When an activity modifies a graph element, a copy of that element is made to enable rollbacks.
- *Aborted activities*: When an activity is aborted, the computational work performed up to that point by that activity is wasted. Furthermore, the runtime system needs to take corrective action to roll back the activity, which adds to the overhead.
- *Dynamic assignment of work*: Threads go to the centralized work-list to get work. This requires synchronization; moreover, if there are many threads and the computation performed in each activity is small, contention between threads will limit speedup.

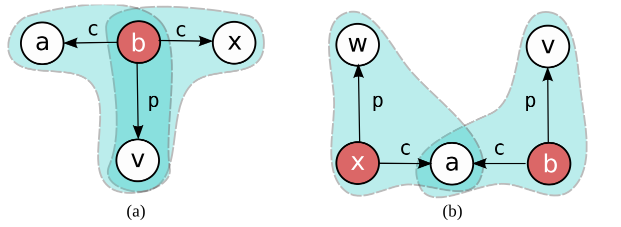


Figure 9. Conflicts occur when two neighborhoods contain the same node.

5. Exploiting structure to optimize parallel execution

Reducing the overheads of the baseline system requires exploiting algorithmic structure in general. For example, when the graph can be partitioned between the threads, abstract locks can be assigned to partitions rather than to graph elements, and separate work-lists can be used for each partition; this reduces the number of aborted activities, contention between threads for the work-list and the overhead of locking [20]. However, partitioning a graph is an overhead in itself. Fortunately, constraint graph rewriting has structure of its own that can be exploited to reduce or even completely eliminate most of these overheads.

5.1 Optimizations

Eliminating abstract locks: To reduce the overhead of enforcing neighborhood constraints, we observe first that if the constraint graph were a read-only data structure, we would not need any abstract locks to ensure commutativity of graph API operations. Unfortunately, the rewrite rules of Figure 6 do update the constraint graph, so this simple optimization cannot be used. However, notice that these rewrite rules never *remove* nodes or edges from the graph; they can only *add* edges to the graph. This means that the two edges in the precondition of each rewrite rule will exist permanently in the graph regardless of what other rewrites are performed, so it is not necessary to use abstract locks to ensure this. As a consequence, when executing a rewrite rule, we need to acquire an abstract lock only for the node at which the edge is added. This reduces the number of abstract lock acquisitions and releases by two-thirds.

While this optimization is relatively straight-forward, a more careful analysis shows that we do not need an abstract lock even at the node at which the edge is added. The only concurrent activities that can happen when an activity I is adding a copy or points edge (a, v) are the following.

- Another activity reads an edge that starts at node a. This edge cannot be removed by I.
- Another activity adds an edge (a, w) such that $v \neq w$. This new edge cannot affect the update rule executed by I, because it does not depend on that edge.

- Another activity tries to add the *same* edge (a, v). The two rules can be interleaved in any fashion and the final state will be the same, provided the concrete representation of the edge set is properly synchronized to ensure that the edge is added only once. The work performed by one of the activities is redundant, but it is irrelevant which one actually performs the addition of the edge.

It is important to note that the elimination of abstract locks depends critically on the fact that the operators (rewrite rules) are *extensive*[3]: they only add edges, and never remove nodes or edges from the graph[4].

Eliminating copying of data: Since there are no conflicts between concurrently executing rewrite rules, there are no rollbacks, so it is unnecessary to make backup copies of modified data.

Aborted activities: There are no aborted activities. Some activities may perform a small amount of redundant work since the edge they want to add may have been added by a concurrent activity, which cannot happen in a sequential implementation.

Therefore, the remaining overhead is the dynamic assignment of work. Since each activity performs a relatively small computation, the overhead of adding and removing work from the centralized work-list can be substantial. To reduce this performance penalty, we use *iteration coalescing* [23], which can be viewed as a data-centric version of loop chunking [27]. When an activity adds an edge to the graph, it checks to see if the new edge violates any invariants at the source node. If so, it puts the work on a local work-list of its own, and performs those activities itself, rather than putting them on the global work-list. Intuitively, the optimization increases the granularity of the work and reduces the performance impact of having a global work-list.

5.2 Discussion

In our current implementation of inclusion-based points-to analysis, the optimizations described in this section are implemented by hand. However, we believe it should be possible to automate these optimizations. The key observation is that the rewrite rules of Figure 6 are extensive operators, and this is straight-forward for a static analysis to deduce from the Galois code since this code uses the Galois graph API, and properties of API methods are available to the static analysis. Note that this kind of analysis would be almost impossible if the code were written in a language like C without using data abstractions.

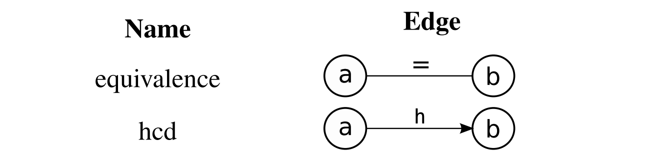


Figure 10. Additional edge types to support merging

A second issue is whether it is worth implementing this analysis and these optimizations in a general-purpose system. In other words, how common are extensive operators in practice? Notice that any flow-insensitive dataflow analysis involves monotone and extensive operators, so these optimizations are useful for all such analyses.

6. Node coalescing

Fahndrich *et al* [7] introduced a key optimization of the basic algorithm for inclusion-based points-to analysis described in Section 2. They observed that the constraint systems of many programs can be reduced substantially by eliminating cycles of copy constraints. For example, a pair of statements of the form *a = b; b = a;* produces a cycle of constraints that imply that *pts(a) = pts(b)*. We call these *equivalent* nodes; coalescing equivalent nodes into a single node reduces the size of the problem. Unfortunately, cycles can also arise dynamically during the constraint process, so we cannot find all equivalent nodes during preprocessing.

In the literature, cycle detection comes in two flavors: *offline* methods [29] look for cycles during a preprocessing phase, while *online* methods [7] look for cycles during the constraint solving process. Some intermediate techniques, such as Hybrid Cycle Detection [8] (HCD), combine the two: potential cycles are identified in the offline phase, and these are collapsed during analysis. Potential cycles arise from statements of the form **a = b; b = *a;*. Without knowing *pts(a)*, we do not know the nodes that participate in cycles with *b*, so these cycles cannot be eliminated during preprocessing, but we can remove them during the constraint solving process whenever we add nodes to *pts(a)*. We experimented with these alternatives, and ultimately settled on combining offline techniques [9] with HCD. Our experiments, which are described in more detail in Section 7, showed that this is faster than online cycle detection even for serial execution; moreover, HCD is easier to parallelize, making it the preferred choice for a parallel implementation.

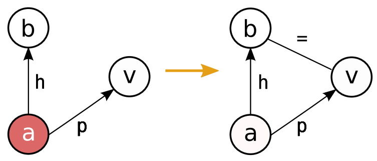
In this section, we focus on the parallelization of the online phase of HCD, which coalesces nodes that must have the same point-to sets. In principle, all that needs to be done is to replace the two nodes being coalesced with a single node that inherits all the edges incident on the eliminated nodes. However, removing nodes from a graph in a parallel implementation can be expensive, so we perform this operation

[3] In mathematics, if D is a partially ordered set, $f : D \to D$ is extensive if $x \leq f(x)$.

[4] If the baseline system used commutativity of method invocations to detect conflicts [18], the system would correctly report that concurrent applications of the rewrite rules do not conflict with each other. However, checking commutativity of graph API method invocations is expensive in general. To eliminate these checks for this particular problem, the system needs global information that all graph API method invocations in the body of the operator commute, which is similar to what is needed when conflicts are checked using exclusive abstract locks.

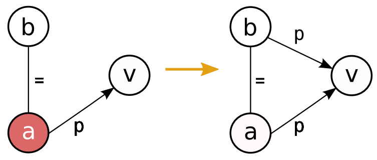
Invariant **Rewrite rule**

$$\exists a \xrightarrow{p} v \wedge \exists a \xrightarrow{h} b \Rightarrow \exists b \overset{=}{\text{---}} v$$



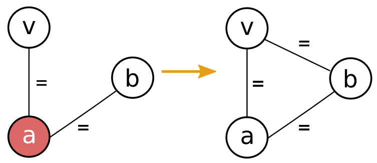
(a) *detect* rule

$$\exists a \xrightarrow{p} v \wedge \exists a \overset{=}{\text{---}} b \Rightarrow \exists b \xrightarrow{p} v$$



(b) *push* rule

$$\exists a \overset{=}{\text{---}} b \wedge \exists a \overset{=}{\text{---}} v \Rightarrow \exists b \overset{=}{\text{---}} v$$



(c) *transitive* rule

Figure 11. Rewriting rules to support node coalescing

implicitly in a way that permits coalescing to happen in parallel with graph rewriting for constraint solving. A simple scheme that accomplishes this is described in Section 6.1. Our actual implementation is a refinement of this scheme, as described in Section 6.2.

6.1 Rewrite rules for equivalent nodes

We start by adding two new types of edges, shown in Figure 10. An *hcd* edge $a \xrightarrow{h} b$ indicates that b and all the variables in $*a$ can be coalesced; these edges are added by the HCD preprocessing phase. An *equivalence* edge such as $a \overset{=}{\text{---}} b$ indicates that the points-to sets of a and b are equal, so they can be coalesced.

The analysis graph is initialized as in the basic algorithm in Section 3, except that hcd edges are added by the offline detection phase of HCD. This initial graph contains no equivalence edges. The set of basic graph rewrite rules in Figure 6 is augmented with three additional rules, shown in Figure 11.

- The *detect* rule implements the HCD cycle detection policy: if there is an hcd edge between a and b, then every variable v pointed by a is equivalent to b.

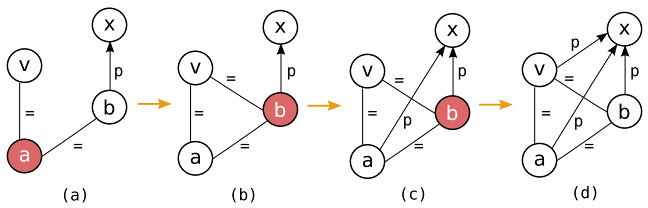


Figure 12. Node collapsing: example.

- The *push* rule states that if there is a points edge $a \xrightarrow{p} v$, and a is equivalent to b, then there must be a points edge $b \xrightarrow{p} v$.
- The *transitive* rule ensures the correct propagation of equivalences throughout the analysis graph. It exploits the transitive nature of the given equivalence: if a is equivalent to b, and b is equivalent to c, then a is equivalent to c.

While the detection rule is particular to HCD, the push and transitive rules are applicable to the coalescing phase of any other algorithm that exploits cycles.

Figure 12 shows how an initial constraint graph is rewritten by application of the new rewriting rules. First, we apply the transitive rule so $\{a, b, v\}$ are in the same equivalence class (Figure 12b). Then we repeatedly apply the push rule to enforce the requirement that the points-to sets of equivalent variables are identical, thereby adding the edges $a \xrightarrow{p} x$ and $v \xrightarrow{p} x$.

6.2 Parallelization and optimizations

Like the constraint graph rewriting rules of Section 3, the new rewrite rules are extensive operators and do not delete nodes or edges from the graph. Therefore, we can apply the same reasoning as in Section 5 to devise a parallel implementation of the extended algorithm that does not require any abstract locking.

On the other hand, this baseline merging scheme has the disadvantage that a points edge is propagated to all equivalent nodes. A better approach is to implement a union-find data structure [34] to track equivalent nodes, and propagate points edges only to representative nodes wherever possible. In our implementation, the union-find data structure is overlayed with the constraint graph by making equivalence edges directional (these are called *representative* edges). The representative for a set of equivalent nodes is the node with the highest ID. Because of concurrent graph rewriting, a node may have several outgoing representative edges. These edges are kept sorted by the node IDs of the destination nodes of these edges; when propagating points edges, we only propagate edges along the first edge in this sequence. A detailed description of a concurrent union-find data structure implemented along these lines will appear elsewhere.

7. Experimental evaluation

Our implementation of the parallel pointer analysis is written in Java. To provide a reasonable comparison with an existing state-of-art analyzer, our code is structured along the lines of the C++ serial implementation of [9], which is available at `http://www.cs.ucsb.edu/~benh/`. For the rest of this section, we will refer to this C++ implementation as the *reference* implementation.

7.1 Key data structures

The points-to algorithm uses two key data structures for the representation of the constraint graph: the Binary Decision Diagram [5] (BDD), used for storing the points-to edges, and the sparse bit vector, used for storing all the other types of edges. In addition to concurrent versions of these data structures, we implemented a thread-safe version of the work-list.

A BDD is a rooted acyclic directed graph. We represent the graph nodes using objects, which are stored in a concurrent hash table. Another concurrent hash table is used for implementing the cache that stores the results of previous operations on the BDD, as it is done in the popular BuDDy library [22]. Synchronization on the hash table is achieved by protecting segments of contiguous buckets with a reentrant lock.

Memory space occupied by BDD nodes that are no longer in use is reclaimed by the Java garbage collector, because the hash table has support for *weak* references. In Java, objects that are reachable only by weak references can be collected by the GC, as opposed to objects reachable by at least one *strong* reference. By relying on the automatic memory management provided by the virtual machine, our BDD implementation does not require the use of reference counts, as it is needed in C implementations like BuDDy. Another (non concurrent) BDD package that takes advantage of the JVM's memory management is SableJBDD [28].

Previous studies have highlighted the importance of variable ordering in BDD implementations [4]. Suboptimal *orderings* result in a BDD with a large number of nodes, consuming more memory and slowing down set operations. In this work, we used the same variable ordering as the one used by the reference implementation.

A sparse bit vector is another standard representation used for the compact encoding of a (sparse) set of elements. In our case, the different sets of outgoing edges associated with each node are represented using sparse bit vectors based on concurrent doubly linked lists. Each node in the linked list contains a double word that can store up to 64 elements on it.

The work-list is represented using a concurrent vector. We use a *dual* [24] work-list, which is divided into two sections: current and next. Active nodes are selected from the current section and pushed onto next, and the two are swapped when current becomes empty. The divided work-list not only results on better performance than a single work-list, but also in less contention: a thread that wants to add an element to a work-list does not have to wait for threads that are getting elements from the work-list. In fact, retrievals could proceed without synchronization, given that the current work-list size is known when the two work-lists are swapped: we could simply assign each thread to a certain index range in the work-list. However, this approach does not perform well in practice because it results on poor load balancing.

As in other unordered algorithms, any order of processing active nodes is legal but the amount of computation performed by different orders may be different. Our implementation uses the same LRF (Least Recently Fired) strategy as the reference implementation. This scheduling policy gives priority to nodes processed furthest back in time, thereby promoting a breadth-first propagation of information through the graph. Although the Java library provides concurrent sets based on skip lists, our experiments showed that a simpler alternative performs better in practice: we keep an atomic flag for every node in the graph to indicates whether it is in the work-list or not (so the work-list becomes a work-set), and the LRF order is enforced by sorting the next work-list at swap time.

	Variables		**Constraints**	
Program	initial	reduced	initial	reduced
perl	53,358	6,031	68,645	10,926
nh	97,933	21,943	114,459	38,178
gcc	120,867	20,053	156,276	32,193
vim	246,941	22,420	108,271	37,422
python	92,596	23,595	111,531	43,534
svn	107,705	30,482	139,848	59,939
gdb	232,811	51,044	241,592	84,238
pine	612,913	60,657	315,900	120,840
php	339,535	59,960	325,891	108,156
gimp	558,864	153,231	649,590	284,252

Figure 13. Benchmark suite: number of variables and constraints.

7.2 Experimental setup

Figure 13 shows the benchmark suite used in our experiments. It consists of ten C programs ranging from 53K-558K variables and 68K-649K constraints. Most of the programs in our benchmark suite have been used by other researchers in this area [9, 25].

The input programs are parsed using the LLVM compiler to generate the initial variables and constraints. In Fig 13, the numbers of variables and constraints are shown before and after offline analysis. In our experiments, we use the three offline algorithms available in the reference implementation: HVN, HRU [9], and HCD. The offline phase results on a constraint graph at least 50% smaller than the original.

The machine used in our experiments is a Sun Fire X2270 (Nehalem server) running Ubuntu Linux version 8.06. The

		perl				**nh**			
HCD	**LCD**	offline	online	total	impr	offline	online	total	impr
no	no	328	1,280	1,608	1.0	416	924	1,340	1.0
yes	no	340	88	428	3.8	464	204	668	2.0
no	yes	328	208	536	3.0	416	244	660	2.0
yes	yes	340	96	436	3.7	464	240	704	1.9

		gcc				**vim**			
HCD	**LCD**	offline	online	total	impr	offline	online	total	impr
no	no	664	224	888	1.0	420	4,444	4,864	1.0
yes	no	712	108	820	1.1	489	792	1,281	3.8
no	yes	664	256	920	1.0	420	808	1,228	4.0
yes	yes	712	276	988	0.9	489	916	1,405	3.5

		python				**svn**			
HCD	**LCD**	offline	online	total	impr	offline	online	total	impr
no	no	684	19,953	20,637	1.0	700	52,107	52,807	1.0
yes	no	728	1,788	2,516	8.2	772	1,708	2,480	21.3
no	yes	684	1,392	2,076	9.9	700	2,324	3,024	17.5
yes	yes	728	1,152	1,880	11.0	772	1,536	2,308	22.9

		gdb				**pine**			
HCD	**LCD**	offline	online	total	impr	offline	online	total	impr
no	no	1,896	136,909	138,805	1.0	1,748	149,193	150,941	1.0
yes	no	2,020	4,084	6,104	22.7	1,904	6,428	8,332	18.1
no	yes	1,896	4,884	6,780	20.5	1,748	6,752	8,500	17.8
yes	yes	2,020	3,372	5,392	25.7	1,904	5,476	7,380	20.5

		php				**gimp**			
HCD	**LCD**	offline	online	total	impr	offline	online	total	impr
no	no	3,520	322,644	326,164	1.0	10,761	743,542	754,303	1.0
yes	no	3,728	4,816	8,544	38.2	11,245	8,593	19,838	38.0
no	yes	3,520	10,941	14,461	22.6	10,761	15,305	26,066	28.9
yes	yes	3,728	5,392	9,120	35.8	11,245	8,477	19,722	38.2

Figure 14. Reference implementation: analysis times, in ms.

system contains two quad-core 2.93 GHz Intel Xeon processors. The 8 CPUs share 24 GB of main memory. Each core has two 32 KB L1 caches and a unified 256 KB L2 cache. Each processor has an 8 MB L3 cache that is shared among the cores. The Java Virtual Machine is the 32-bit Sun HotSpot server virtual machine version 1.6.0_20.

7.3 Justifying the use of HCD

Our first set of experiments evaluates the performance of the (sequential) reference implementation using the HCD and LCD approaches. The objective is to justify the choice of the HCD algorithm for the parallel implementation.

The analysis times (in milliseconds) are shown in Figure 14. Each benchmark was run three times, and the average runtime is reported. For every benchmark in the input suite, we evaluated the effect of the two online optimizations present in the reference implementation: HCD and LCD [8]. All the four possible combinations obtained by enabling/disabling these techniques are shown in the table of results.

The analysis time is divided into two columns. The *offline* column refers to the time spent in the HVN, HRU, and HCD (if applicable) offline algorithms. The *online* column refers to the time consumed by constraint solving. The *total* column is the sum of the offline and online runtimes. We omitted the time spent loading the constraints from disk and initializing the BDD (these consume less than 10% of the total analysis time).

To facilitate the comparison between the different configurations we show a $impr$ (improvement) column, which is the result of dividing the total runtime with no optimizations (HCD=no, LCD=no) by the total runtime for the given configuration. Higher values of $impr$ indicate better performance. For example, in the case of the perl benchmark we get $impr = 1608/428 = 3.8$ by using only HCD, and $impr = 1608/536 = 3.0$ by using only LCD.

The results show the following.

- The application of any of the cycle detection techniques drastically reduces the overall analysis time for all the

	perl			nh			gcc			vim			python		
threads	offline	online	total	offline	online	total	offline	online	total	offline	online	total	offline	online	total
1	199	144	343	142	273	415	466	140	606	283	1,083	1,366	158	2,593	2,751
2	199	136	335	142	230	372	466	104	570	283	730	1,013	158	1,575	1,733
4	199	130	329	142	210	352	466	110	576	283	491	774	158	1,157	1,315
6	199	130	329	142	199	341	466	118	584	283	431	714	158	947	1,105
8	199	131	330	142	222	364	466	135	601	283	428	711	158	908	1,066

	svn			gdb			pine			php			gimp		
threads	offline	online	total	offline	online	total	offline	online	total	offline	online	total	offline	online	total
1	480	2,490	2,970	1,144	6,153	7,297	421	8,129	8,550	1,959	7,467	9,426	6,584	13,716	20,300
2	480	1,928	2,408	1,144	4,866	6,010	421	5,752	6,173	1,959	5,559	7,518	6,584	11,752	18,336
4	480	1,409	1,889	1,144	3,392	4,536	421	4,239	4,660	1,959	3,956	5,915	6,584	8,753	15,337
6	480	1,238	1,718	1,144	2,895	4,039	421	4,019	4,440	1,959	3,654	5,613	6,584	7,745	14,329
8	480	1,153	1,633	1,144	2,683	3,827	421	3,498	3,919	1,959	3,582	5,541	6,584	7,170	13,754

Figure 15. Parallel implementation: analysis times, in ms.

inputs. This effect is more evident for large benchmarks. For instance, gimp takes 12 minutes to complete in the absence of any optimization, but when LCD or HCD are applied, the analysis finishes in less than 30 seconds.

- We can achieve nearly optimal runtimes just by applying HCD. The addition of Lazy Cycle Detection results in improvements in performance for half of the benchmarks, but the gain is very small except in the case of python.

These results justify the choice of HCD for node coalescing in our parallel implementation.

7.4 Results of parallel implementation

Figure 15 show the analysis times (in milliseconds) for each program in the benchmark suite, and different numbers of threads. To account for the effects of JIT compilation, each benchmark was run nine times, and the average runtime is reported. We verified the output (points-to of every variable in the original program) after the first run against the reference implementation. Finally, we minimize the influence of garbage collection by maximizing the size of the heap used by the JVM.

As in Figure 14, the analysis time is divided into two phases. Because the offline phase is sequential, the time spent on it does not change for different numbers of threads. The *online* column refers to the time taken by our parallel implementation of constraint solving, which uses HCD but not LCD.

The first observation about the results in Figure 15 is that the analysis times are not always proportional to the input size. This also applies to the reference implementation. For instance, the number of variables and constraints in the python benchmark is similar to that of vim, but the analysis of python takes twice as much time to complete. Clearly the structure of the original constraint graph plays a fundamental role in the analysis time. Sparse constraint graphs demand little points-to propagation, so the algorithm will converge rapidly.

Figure 15 shows the runtimes of our parallel implementation on different numbers of threads. The scaling of a parallel program is the execution time for one thread divided by the execution time for x threads. The scaling achieved is dependent on the amount of computation that needs to be done, and tends to improve when the one-thread execution times get bigger. For small benchmarks (perl, nh and gcc) the performance degrades when we use many threads, since the amount of computation to be done is so small -the online analysis of nh finishes within 273ms with one thread- that the overheads of parallelization dominate the total runtime. For all the other inputs we achieve a good scaling by using two threads, although the best total runtime is achieved with eight threads. It is interesting to see that the best scaling (2.85x) is obtained for python, while for our biggest input program (gimp) the scaling is almost 2x.

Because we only parallelized the constraint solving phase of the analysis, the offline section now becomes a performance bottleneck. For example, the offline phase of php represents the 20% of the total runtime if we use one thread, but the ratio increases to 35% with eight threads. We leave the parallelization of the offline phase as future work.

7.5 Comparative study

We now describe a comparative study of the results obtained for the parallel and reference implementations, using only HCD. While we configured the two implementations so they use the same online and offline techniques, our analysis (as the rest of the Galois framework) is written in Java, while Hardekopf's implementation is written in C++. Therefore, establishing a completely fair comparison is extremely difficult: one language is interpreted, the other is compiled; the reference implementation uses the BuDDy BDD library, while we implemented a pure Java BDD package that relies on the garbage collector for eliminating dead nodes, etc.

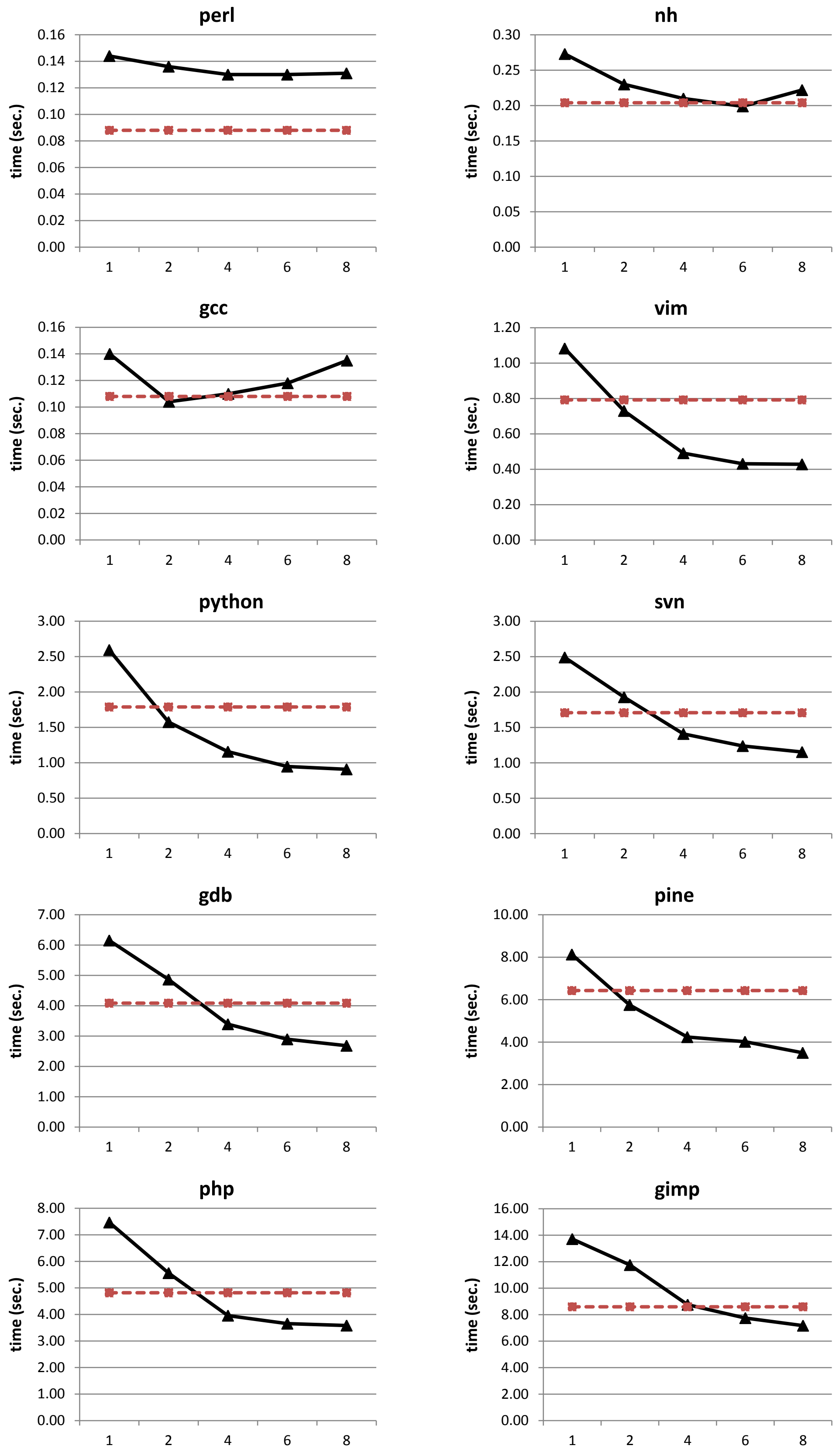


Figure 16. Online analysis times, in seconds: parallel (▲) vs. reference (🟥) implementation.

	REFERENCE			PARALLEL		
input	offline	online	total	offline	online	total
perl	340	88	**428**	199	130	**329**
nh	464	204	**668**	142	199	**341**
gcc	712	108	**820**	466	104	**570**
vim	489	792	**1,281**	283	428	**711**
python	728	1,788	**2,516**	158	908	**1,066**
svn	772	1,708	**2,480**	480	1,153	**1,633**
gdb	2,020	4,084	**6,104**	1,144	2,683	**3,827**
pine	1,904	6,428	**8,332**	421	3,498	**3,919**
php	3,728	4,816	**8,544**	1,959	3,582	**5,541**
gimp	11,245	8,593	**19,838**	6,584	7,170	**13,754**

Figure 17. Best total analysis times, in ms.

Nevertheless, the value of comparing the runtimes of the two analyses lies then in showing that our parallel implementation is competitive with a highly tuned, state-of-the-art C++ implementation; in most cases, our parallel Java implementation achieves a speed-up over the optimized C++ sequential implementation.

We plotted the overall runtime for the two analyses in Figure 16. Each dark triangle marker (▲) in a plot corresponds to the online runtime of our analysis for that particular combination of benchmark and number of threads (*online* column in Figure 15). Each bright square marker (🟥) in a plot corresponds to the online runtime of the reference implementation for that particular benchmark in the HCD=yes, LCD=no row of Figure 14.

The overheads of the parallel analyzer can be observed when only one thread is used: our implementation can be twice as slow as the reference program, even though our (sequential) offline phase usually takes less time to complete. The benefits of using a parallel implementation become clear for larger thread counts since we usually achieve a significant speedup over the C++ implementation.

Figure 17 provides a comparison between the total analysis times for the parallel and reference implementations. The times shown for the parallel version correspond to the fastest execution. The best speedup achieved for the whole analysis is 2.5x for the python benchmark (note that Figure 16 shows only the online analysis times).

7.6 Memory consumption

Figure 18 shows the memory consumption (in megabytes) of the parallel and reference implementations among the ten input benchmarks. The numbers in the table correspond to the maximum amount of memory required by each analyzer. We measured the memory usage of the Java (parallel) program using the tools provided by the Hot Spot virtual machine; the reference implementation uses the profiler available in the Google Performance Tools [31].

The memory consumption of our code is significantly higher (a factor of 1.5-2.5x). The BDD and the sparse bit

input	REF	PAR
perl	309	340
nh	356	320
gcc	356	280
vim	368	520
python	356	550
svn	347	542
gdb	439	730
pine	972	1,167
php	529	1,120
gimp	732	1,750

Figure 18. Memory usage, in megabytes.

vectors dominate the memory usage of both applications. Since we are using the same data structures (but concurrent versions) and cache policies as the reference implementation, we believe these differences might arise from the fact that our implementation is in Java, while the reference implementation is in C++.

8. Conclusions and future work

This paper describes the first parallel implementation of a state of the art inclusion-based points-to analysis. Irregular algorithms of this kind are very difficult to parallelize because dependences between computations are functions of runtime data, so speculation is needed in general to exploit the parallelism. However, the overheads of speculation can be reduced dramatically by exploiting algorithmic structure. In our experience, reasoning about parallelism in terms of the operator formulation of algorithms greatly simplifies the task of devising an efficient version of the algorithm. The operator formulation exposed algorithmic structure that we exploited in optimizations to reduce the overheads of speculative execution.

Our parallelization of Andersen's algorithm exploits the fact that the state of the analysis grows monotonically during the execution of the algorithm; tree-building algorithms such as Prim's MST algorithm and n-body simulation algorithms also have this structure, which can be exploited for efficient parallel implementation.

The experimental results in Section 7 confirm that our approach is not only feasible, but can also be competitive with highly tuned sequential implementations of pointer analysis. Further improvements require attention to parallelizing the offline component of the analysis.

Our implementation depends on two data structures: the hash table used for representing the BDD, and the doubly linked list used for representing a sparse bit vector. The concurrent versions of these two popular data types use locking (mutexes). It would be interesting to study the performance of the analysis when lock-free implementations [11, 33] of the two data structures are used.

Acknowledgments

The authors would like to thank Ben Hardekopf for generously providing the benchmarks used in our experiments and clarifying many aspects of his implementation.

References

[1] Galois website. `http://iss.ices.utexas.edu/galois/`.

[2] A. Aho, R. Sethi, , and J. Ullman. *Compilers: principles, techniques, and tools*. Addison Wesley, 1986.

[3] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[4] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114, New York, NY, USA, 2003. ACM.

[5] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[6] H. Ehrig and M. Löwe. Parallel and distributed derivations in the single-pushout approach. *Theoretical Computer Science*, 109:123–143, 1993.

[7] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1998. ACM.

[8] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, 2007.

[9] Ben Hardekopf and Calvin Lin. Exploiting pointer and location equivalence to optimize pointer analysis. In *SAS*, pages 265–280, 2007.

[10] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: a million lines of c code in a second. *SIGPLAN Not.*, 36(5):254–263, 2001.

[11] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216, New York, NY, USA, 2008. ACM.

[12] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[13] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM.

[14] Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 249–259, New York, NY, USA, 2008. ACM.

[15] Ken Kennedy and John Allen, editors. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann, 2001.

[16] J. W. Klop, Marc Bezem, and R. C. De Vrijer, editors. *Term Rewriting Systems*. Cambridge University Press, New York, NY, USA, 2001.

[17] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 48(9):866–880, 1999.

[18] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not. (Proceedings of PLDI 2007)*, 42(6):211–222, 2007.

[19] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, New York, NY, USA, 2009. ACM.

[20] Milind Kulkarni, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. Optimistic parallelism benefits from data partitioning. *SIGARCH Comput. Archit. News*, 36(1):233–243, 2008.

[21] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[22] Jorn Lind-Nielsen. Buddy, a Binary Decision Diagram package. `http://www.itu.dk/research/buddy/`. Department of Information Technology, Technical University of Denmark.

[23] Mario Mendez-Lojo, Donald Nguyen, Dimitrios Prountzos, Xin Sui, M. Amber Hassaan, Milind Kulkarni, Martin Burtscher, and Keshav Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 3–14, January 2010.

[24] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[25] Fernando Magno Quintao Pereira and Daniel Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09: Proceedings of the 2009 International Symposium on Code Generation and Optimization*, pages 126–135, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Keshav Pingali, Milind Kulkarni, Donald Nguyen, Martin Burtscher, Mario Mendez-Lojo, Dimitrios Prountzos, Xin Sui, and Zifei Zhong. Amorphous data-parallelism in irregular algorithms. regular tech report TR-09-05, The University of Texas at Austin, 2009.

[27] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput.*, 36(12):1425–1439, 1987.

[28] Feng Qian. SableJBDD, a Java Binary Decision Diagram Package. `http://www.sable.mcgill.ca/~fqian/SableJBDD/`.

[29] Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 47–56, New York, NY, USA, 2000. ACM.

[30] Erik Ruf. Partitioning dataflow analyses using types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 15–26, New York, NY, USA, 1997. ACM.

[31] Craig Silverstein. Google Performance Tools. `http://code.google.com/p/google-perftools/`.

[32] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM.

[33] Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *J. Parallel Distrib. Comput.*, 68(7):1008–1020, 2008.

[34] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[35] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM.

[36] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: a step toward practical analyses. *SIGSOFT Softw. Eng. Notes*, 21(6):81–92, 1996.